Parallel design of sparse deep belief network with multi-objective optimization

Li, Yangyang; Fang, Shuangkang; Bai, Xiaoyu; Jiao, Licheng; Marturi, Naresh

DOI:
10.1016/j.ins.2020.03.084

Document Version
Peer reviewed version

Citation for published version (Harvard):
Li, Y, Fang, S, Bai, X, Jiao, L & Marturi, N 2020, 'Parallel design of sparse deep belief network with multi-objective optimization', *Information Sciences*, vol. 533, pp. 24-42. https://doi.org/10.1016/j.ins.2020.03.084

Parallel Design of Sparse Deep Belief Network with Multi-objective Optimization

Yangyang Li[1]*, Shuangkang Fang[1], Xiaoyu Bai[1], Licheng Jiao[1], and Naresh Marturi[2]

[1]Key Laboratory of Intelligent Perception and Image Understanding of Ministry of Education, International Research Center for Intelligent Perception and Computation, Joint International Research Laboratory of Intelligent Perception and Computation, School of Artificial Intelligence, Xidian University, Xi'an, Shaanxi Province 710071

[2]Extreme Robotics Laboratory, University of Birmingham, Edgbaston, B15 2TT, UK

*yyli@xidian.edu.cn

Abstract

Deep belief network (DBN) is an import deep learning model and restricted Boltzmann machine (RBM) is one of its basic models. The traditional DBN and RBM have numerous redundant features. Hence an improved strategy is required to perform sparse operations on them. Previously, we have proposed our own sparse DBN (SDBN): using a multi-objective optimization (MOP) algorithm to learn sparse features, which solves the contradiction between the reconstruction error and network sparsity of RBM. Due to the optimization algorithm and millions of parameters of the network itself, the training process is difficult. Therefore, in this paper, we propose an efficient parallel strategy to speed up the training of SDBN networks. Self-adaptive Quantum Multi-objectives Evolutionary algorithm based on Decomposition (SA-QMOEA/D) that we have proposed as the multi-objective optimization algorithm has the hidden parallelism of populations. Based on this, we not only parallelize the DBN network but also realize the parallelism of the multi-objective optimization algorithm. In order to further verify the advantages of our approach, we apply it to the problem of facial expression recognition (FER). The obtained experimental results demonstrate that our parallel algorithm achieves a significant speedup performance and a higher accuracy rate over previous CPU implementations and other conventional methods.

Keywords: restricted Boltzmann machine; deep belief network; multi-objective optimization; parallel acceleration; facial expression recognition; GPU

1. Introduction

In recent years, deep learning has attracted significant attention from both academia and industry due to its ability to boost performance in various computer vision applications. The fast layer-wise training algorithm proposed by Hinton et al. [1], where multiple restricted Boltzmann machines (RBM) are stacked and trained in a greedy manner to form deep belief networks (DBN), ameliorated the learning ability of deep learning algorithms for image recognition, natural language processing, motion capture etc. The DBN model, which is considered to be one of the most effective deep learning algorithms, can learn a complex

nonlinear model with millions of parameters from unlabeled data [2-3]. However, with large number of parameters, the model may produce redundant features without any constraints. In such cases, to learn more abstract features, some previous studies have added a regularization term on the hidden units [4-8]. In addition to constraining the hidden units to be sparse, we also need to ensure that the reconstruction error is minimized to learn more useful representation from inputs. Nevertheless, this will inevitably lead to compressed representation and loss of information [9]. When taken together, the reconstruction error and the regularization term are in conflict. Dealing with it, previously we have proposed a sparse DBN (SDBN) based on multi-objective optimization (MOP) to avoid the problem of parameter selection [10].

When using MOP algorithms to optimize a DBN, the choice of objective function is crucial. It requires both a good representation of the problem to be optimized and an implementation that cannot be overly complex. To this extent we have selected the Kullback-Leibler divergence (KL divergence) [11] and the L1 norm of hidden units [12] as our objective functions. The underlying reason for selecting KL divergence is that it can show the proximity of two probability distributions, which can measure the distribution error between the input data and the reconstructed data in RBM. Similarly, L1 norm of hidden units has been selected to consider the balance between the implementation of subsequent parallel algorithms and the optimization effect. Usually, sparse representation is represented by L0 norm minimization. However, solving it is a NP-hard problem [13], which requires more iterations to find a feasible solution. Also, since the activation value of the hidden units is non-zero under the action of the sigmoid activation function, the L0 norm is always constant. In contrast, the L1 norm can generate sparse coefficients and is robust to uncorrelated features [14-15].

For solving MOP problems, multi-objective evolutionary algorithm (MOEA) [16] is a common choice. Multi-objective evolutionary algorithm based on decomposition (MOEA/D) [17] is an efficient MOEA algorithm (based on mathematical programming) that has the advantages of fast convergence and good distribution performance. In 2015, Gong et al. proposed a self-adaptive multi-objective evolutionary algorithm based on decomposition (SA-MOEA/D) for sparse feature learning [9]. However, one main drawback of this method is it is very slow to converge to an optimal value. In this regard, we introduce a quantum mechanism based on SA-MOEA/D to increase population diversity, which can speed up the convergence of the algorithm and improve its search ability. Also, with this, the major advantage is it is easier to design parallel algorithms. We call this new algorithm as self-adaptive quantum multi-objectives evolutionary algorithm based on decomposition (SA-QMOEA/D). Although the MOP algorithm can significantly improve the performance of a DBN, it greatly increases the difficulty of network training [18]. A common handling strategy is to accelerate the training process in parallel. However, most of the conventional methods only focus on speeding up the complex matrix operations in the network and are inattentive to the optimization algorithm.

Designing a parallel algorithm that considers both rapid computations and network

optimization is an extremely difficult process due to aforementioned reasons. Also, it has to be noted that not all optimization algorithms can demonstrate high performance merely by following a parallel architecture. In this context, we propose a method to parallelize the SDBN using graphics processing unit (GPU). The proposed parallel acceleration algorithm accelerates both the SDBN network and the SA-QMOEA/D optimization algorithm. The main reason for selecting a GPU as our accelerating device is that it is suitable for large-scale parallel computing on a two-level hierarchy of blocks and threads [19]. Alternative works regarding parallel implementations, e.g. using FPGA [21] and Hadoop framework [22], can be found in [18-20]. In case of FPGA or Hadoop, implementations usually require special (expensive) hardware external to a PC and are not flexible enough. Whereas with GPU, which is generally available as an inbuilt hardware with a modern-day PC, is comparatively inexpensive and can achieve similar accuracy with faster computations. Also, when compared to a conventional CPU, its consumption is very less.

Apart from this, we have analyzed the performance of the proposed method by applying it to the facial expression recognition problem, which is an active and challenging problem in computer vison research. Although a convolutional network with a two-dimensional (2-D) input space has more advantages, its performance can be compared with the conventional methods such as nearest neighbors (NN) [46], support vector machine (SVM) [47-48], sparse representation classification (SRC) [49] etc. Similar to these other methods, we also extract and reduce the dimension of the image features, and the resultant 1-D vector is used as the input for the SDBN. The obtained experimental results using two publicly available datasets demonstrate the efficiency of our proposed approach.

The rest of the paper is organized as follows: Section 2 presents our SRBM and SDBN algorithms. Section 3 provides our parallel implementation of SRBM and SDBN. Section 4 introduces the application of parallel SDBN in FER. Section 5 reports the experimental results. Section 6 draws the conclusion.

2. SRBM and SDBN

Currently, the combination of evolutionary algorithms and neural networks is mostly concerned with the hyperparameter optimization and neural network architecture search. In our SDBN, we construct a multi-objective optimization problem with RBM, and use the evolutionary algorithms to solve it. This is an alternate way of combining evolutionary algorithms and neural networks.

2.1 RBM and DBN

Boltzmann machines were first proposed by Hinton and Sejnowski in 1986 [2]. Later, Paul Smolensky proposed a modified Boltzmann machine, which is also called as a *restricted Boltzmann machine* [23]. In general, RBM is an energy model, which is consisted of two

layers, namely visible and hidden layers. The design architecture of an RBM is shown in Fig. 1.

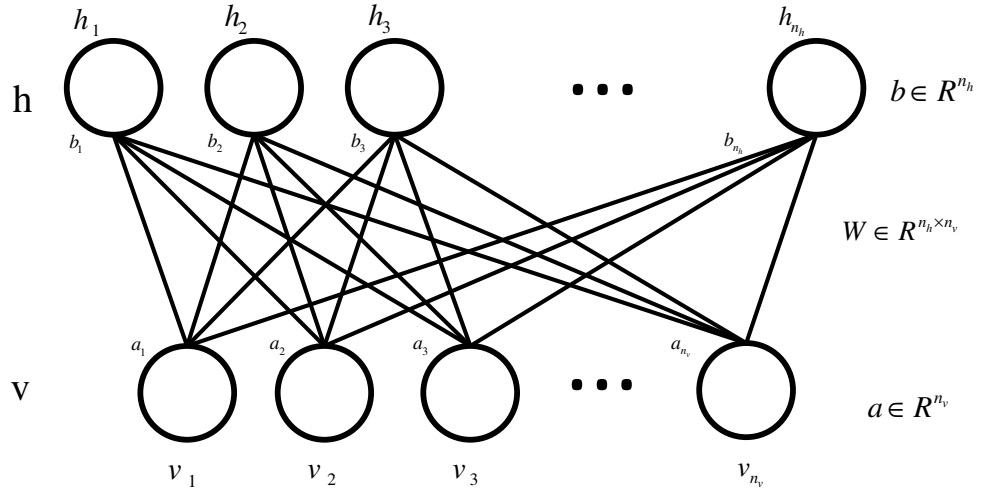


Fig. 1. Design architecture of an RBM showing various nodes in hidden (top) and visible (bottom) layers.

In Fig. 1, the visible layer v is used to input training data whereas the hidden layer h contains feature detectors. n_h and n_v represent the number of hidden and visible units respectively. v_i and h_i denote respectively the i^{th} visible and hidden units. a_i and b_i are the biases. There are no connections between the individual units of the same layer; however, each unit is still fully connected with the units of the other layer by a symmetrical weight matrix W. The energy of visible and hidden units is represented as follows

$$E(\mathrm{v},\mathrm{h}) = -\sum_{i=1}^{n_v}\sum_{j=1}^{n_h} w_{ij} h_i v_j - \sum_{i=1}^{n_v} a_i v_i - \sum_{j=1}^{n_h} b_j h_j \tag{1}$$

The marginal distribution of a visible vector is given by Eq. (2).

$$p(\mathrm{v}) = \sum_h p(\mathrm{v},\mathrm{h}) = \frac{1}{Z}\sum_h e^{-E(\mathrm{v},\mathrm{h})} \tag{2}$$

where, Z is a normalization constant given by Eq. (3).

$$Z = \sum\nolimits_{v,h} e^{-E(\mathrm{v},\mathrm{h})} \tag{3}$$

It is necessary to maximize the likelihood function given by Eq. (2) to learn the value of the RBM parameter $\theta = \{w, a, b\}$ [25-26]. The gradient $\frac{\partial L}{\partial \theta}$ can be calculated as follows.

$$\begin{aligned} \frac{\partial L}{\partial \theta} &= \sum_{v,h} p(\mathrm{v},\mathrm{h}) \partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta} - \sum_h p(\mathrm{h} \mid \mathrm{v}) \partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta} \\ &= \left\langle \partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta} \right\rangle_{P^0} - \left\langle \partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta} \right\rangle_{P_\theta^1} \end{aligned} \tag{4}$$

where $\langle \square \rangle_{P^0}$ represents the expectation of $\partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta}$ under the joint distribution $p(\mathrm{v},\mathrm{h})$. $\langle \square \rangle_{P_\theta^1}$ represents the expectation of $\partial \frac{E(\mathrm{v},\mathrm{h})}{\partial \theta}$ under the conditional distribution $p(\mathrm{h}\,|\,\mathrm{v})$. It is relatively easy to calculate $p(\mathrm{h}\,|\,\mathrm{v})$; however, the computation of $p(\mathrm{v},\mathrm{h})$ is not straightforward. One way to solve this is by using the contrastive divergence (CD) algorithm with one-step Gibbs sampling proposed by Hinton [27]. Following which the gradient given in Eq. (4) is re-written as:

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}} &= \langle v_i h_j \rangle_{P^0} - \langle v_i h_j \rangle_{P_\theta^1} \\
\frac{\partial L}{\partial a_i} &= \langle v_i \rangle_{P^0} - \langle v_i \rangle_{P_\theta^1} \\
\frac{\partial L}{\partial b_j} &= \langle h_j \rangle_{P^0} - \langle h_j \rangle_{P_\theta^1}
\end{aligned} \tag{5}$$

Next, as mentioned before, a DBN is formed by stacking multiple RBMs. Its architecture and the training process are depicted in Fig. 2.

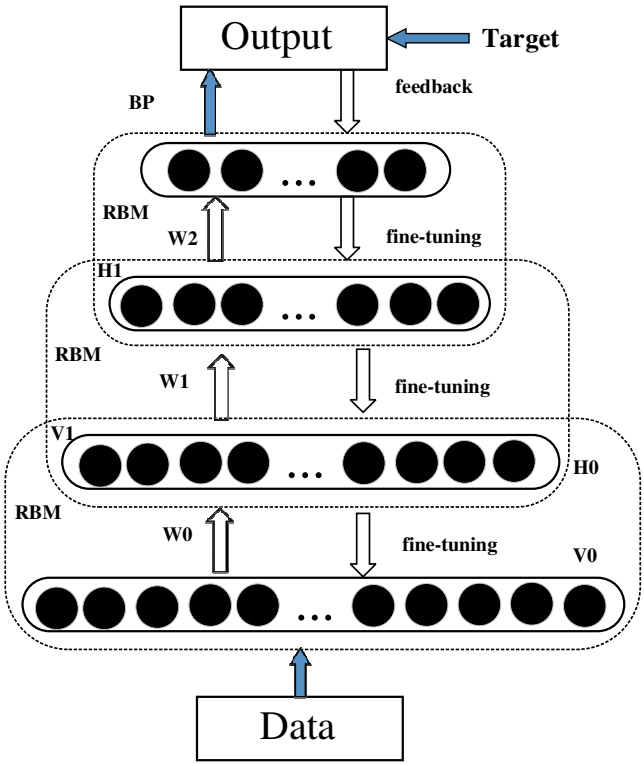


Fig. 2. Architecture and training process of a DBN.

The DBN shown in Fig. 2 is formed by stacking three RBMs where each of them is represented by a dotted box. Its training process is divided into two stages: unsupervised training and fine-tuning. In the first stage, i.e., during unsupervised training, the very first

RBM (bottom one in Fig. 2) is trained using CD algorithm. Later, its hidden layer is used as the visible layer to train the subsequent RBM. This process is repeated for all the remaining RBMs. Next, during the second stage, i.e., fine-tuning, the DBN is fine-tuned by back-propagation (BP) using the parameters obtained from the first stage as initial values [35]. It is worth noting that by initializing the network using the parameters obtained from the first stage instead of random initialization avoids the local optima during training.

2.2 Sparse RBM based on multi-objective optimization

Although a DBN can learn a complex nonlinear model from unlabeled data, with the increased number of parameters, the model may produce redundant features without any constraints. To avoid this, we have proposed the sparse RBM and the sparse DBN based on multi-objective optimization. The multi-objective optimization problem with q objective functions is expressed as:

$$\begin{aligned} &\min\ F(x) = (f_1(x), f_2(x), \cdots, f_q(x))^T \\ &st \quad x \in \Omega \end{aligned} \tag{6}$$

where, Ω represents the feasible region of decision space, $F : \Omega \to R^q$ represents q objective functions, and R^q denotes the objective space. As mentioned previously, in DBN, the reconstruction error and the regularization term are in conflict. To that end, we have selected KL divergence and L1 norm of hidden units as our objective functions. Using which, our cost function $L^*(\theta)$ and objective function $\min F(\theta)$ can be expressed as follows.

$$L^*(\theta) = \mathrm{KL}(P^0 \,\|\, P_\theta^\infty) + \lambda \sum_{l=1}^{T} \| \, p(h^{(l)} \,\|\, v^{(l)}) \, \|_1 \tag{7}$$

$$\min F(\theta) = (f_1, f_2) = (\sum \mathrm{KL}(P^0 \,\|\, P_\theta^\infty), \quad \sum_{l=1}^{T} \| \, p(h^{(l)} \,\|\, v^{(l)}) \, \|_1 \tag{8}$$

where, λ is the regularization parameter, T is the number of layers, P^0 and P_θ^∞ represent the initial and balanced distributions of the data. It is known that maximizing the log likelihood is equivalent to minimizing the KL divergence [1]. Accordingly, the gradient of KL divergence ∇g_{KL} can be obtained as in Eq. (5), while for the L1 regular term, the gradient is computed as follows:

$$-\frac{\partial}{\partial w_{ij}} \sum_{l=1}^{T} \| \, p(h^{(l)} \,\|\, v^{(l)}) \, \|_1 = -\sum_{l=1}^{T} p_j^{(l)} (1 - p_j^{(l)}) v_i^{(l)} \tag{9}$$

$$-\frac{\partial}{\partial b_j} \sum_{l=1}^{T} \| \, p(h^{(l)} \,\|\, v^{(l)}) \, \|_1 = -\sum_{l=1}^{T} p_j^{(l)} (1 - p_j^{(l)}) \tag{10}$$

Because the degree of activation of hidden units is directly controlled by bias, we only update the bias for the sake of simplicity. The learning algorithm of SRBM based on MOP is summarized in algorithm 1.

Algorithm 1 (Learning algorithm of SRBM based on MOP)

Step 1 Initialization:

- Initialize the parameters $\{w, a, b\}$ of SRBM
- Initialize termination conditions

Step 2 Update the parameters:

Step 2.1 calculate gradients:

- Use Eq. (5) to get the gradient ∇g_{KL} of KL divergence
- Use Eq. (9) and Eq. (10) to compute the gradient ∇g_{L1} of L1 regular term

Step 2.2 Update the parameters according to the following rule:

$$\theta := \theta + \varepsilon(\nabla g_{KL} + \nabla g_{L1}),\ \theta \in \{w, a, b\} \tag{11}$$

where, ε is the learning rate

Step 3 MOP

- Use SA-QMOEA/D algorithm (shown in algorithm 2) to get EP
- Randomly select a solution from EP as the new parameters

Step 4 Repeat Steps 2-3 until the termination conditions are satisfied

We use our SA-QMOEA/D algorithm to optimize the objective function in Eq. (8). The two main advantages of this algorithm are the quantum mechanism and the hidden population parallelism [28], which can effectively increase the diversity of the population and accelerate the convergence rate. The details of SA-QMOEA/D algorithm are presented in algorithm 2. It uses Chebyshev decomposition [17] to convert the MOP problem into several scalar optimization problems. Main parameters of our method are listed below:

- N is the number of individuals.
- $[x^1, ..., x^N] \in \Omega$ are all individuals of the population.
- $[\theta^1, ..., \theta^N] \in \Omega_Q$ are quantum chromosomes.
- $FV^i = F(x^i), i = 1, ..., N$ represents the fitness function of each individual.
- $z = (z_1, ..., z_m)$ represents the best value set for every objective function, m is the number of objective functions. For instance, in Eq. (8), z_1 represents the optimal

solution of the objective function f_1.

● $[\lambda^1,...,\lambda^N]$ represent N uniformly spread weight vectors, which can be obtained by the Chebyshev approach [17].

● g^{te} is the subproblem of objective functions [17], which can be defined as follow.

$$g^{te}(x \mid \lambda^j, z^*) = \max_{1 \le j \le m} \left\{ \lambda_j^i \mid f_j(x) - z_j^* \right\} \tag{12}$$

Algorithm 2 (SA-QMOEA/D algorithm)

Step 1 Initialization:

Step 1.1 Initialize N uniformly spread weight vectors $[\lambda^1,...,\lambda^N]$.

Step 1.2 Initialize the field B:

- compute the Euclidean distances between any of the two weight vectors
- select T closest weight vectors $[\lambda^{i1},....\lambda^{iT}]$ for each λ^i and then get $B(i) = (\lambda^{i1},....\lambda^{iT})$

Step 1.3 Initialize the population:

- Initialize the chromosome space $[\theta^1,...,\theta^N] \in \Omega_Q$
- transform Ω_Q to get $[x^1,...,x^N] \in \Omega$
- for each x^i in Ω: calculate $FV^i = F(x^i)$

Step 1.4 Initialize solution $z = (z_1,...,z_m)$ according to FV^i

Step 2 Population evolution:

for each λ_i in $[\lambda^1,...,\lambda^N]$:

- Randomly select three different elements from $B(t)$ to get the corresponding quantum chromosomes. Then, apply the recombination and mutation operator on them to get a new solution y.
- for each z_j in $(z_1,...,z_m)$: if $z_j > f_j(y)$:set $z_j = f_j(y)$.
- for each λ^j in B: if $g^{te}(\mathrm{y} \mid \lambda^j, \mathrm{z}) \le g^{te}(x^j \mid \lambda^j, \mathrm{z})$: $x^j = y$, $FV^j = F(y)$.
- if there is no solution in EP is better than $F(y)$: add $F(y)$ to EP.

Step 3 Repeat Step 2 until the termination condition is satisfied.
Step 4 Output EP

SA-QMOEA/D algorithm can find a balanced optimal solution between the reconstruction error and the regularization term. In other words, the algorithm not only ensures that the RBM learns a complex nonlinear mapping but also guaranties that the network parameters are sparse.

3. Parallel implementation

Although the SRBM algorithm based on MOP can learn sparse features, its training time is high due to the higher number of iterations it takes to find an optimal solution. Furthermore, additional computations are required to find the numerous network parameters. In this section, we use the population implicit parallelism to design the optimization algorithm in parallel, which significantly reduces the SDBN training time.

3.1 Compute unified device architecture - CUDA

In the early days, a GPU has been designed for high-speed graphics that are inherently parallel. It has the architecture that is suitable for highly parallelized computation tasks with less logic control as it can generate several threads in parallel to speed up the process. In order to fully utilize the capabilities of a GPU a specialized software framework is indispensable. The parallel computing platform, CUDA, developed by the Nvidia Corporation is one of the well-known software platforms for general purpose processing as well as for computing and executing programs on GPU [31]. It has been used in this work to implement and execute our parallel algorithms on a GPU.

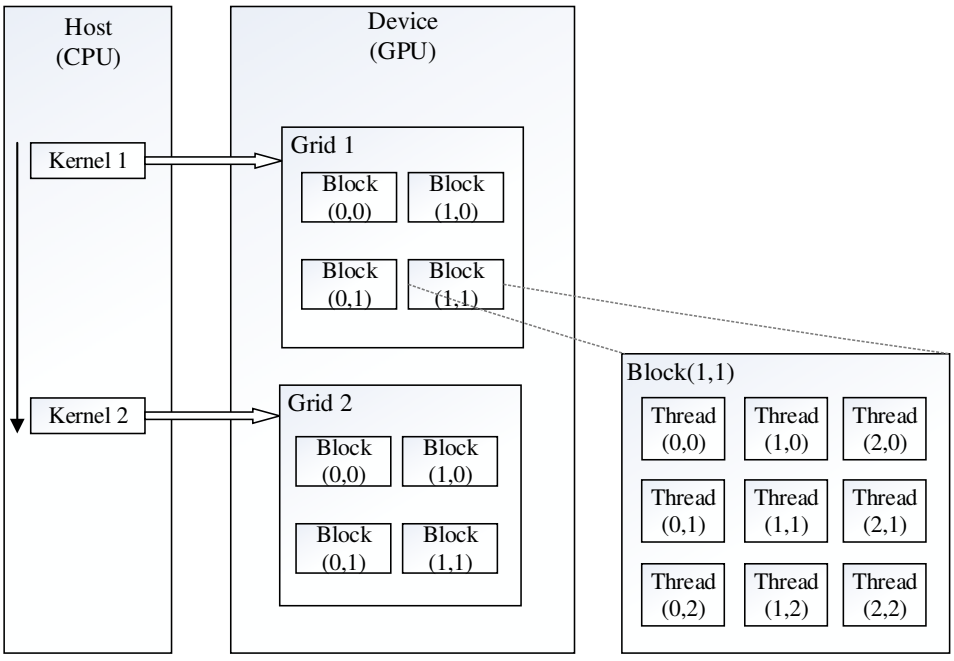


Fig. 3. Illustration of the CUDA programming model.

The programming model of CUDA is shown in Fig. 3. It consists of two modules: host

and device. The serial code is executed on the host, while the parallel code (also called kernel function) is executed on the device [31]. The thread model, i.e., multiple thread blocks are combined to form a grid and each kernel function has several grids. All thread blocks in a grid contain equal number of threads. A thread is the basic execution unit and contains unique block and thread IDs. When using CUDA to execute parallel programs, it is necessary to specify the type of the memory to store data and intermediate computation parameters.

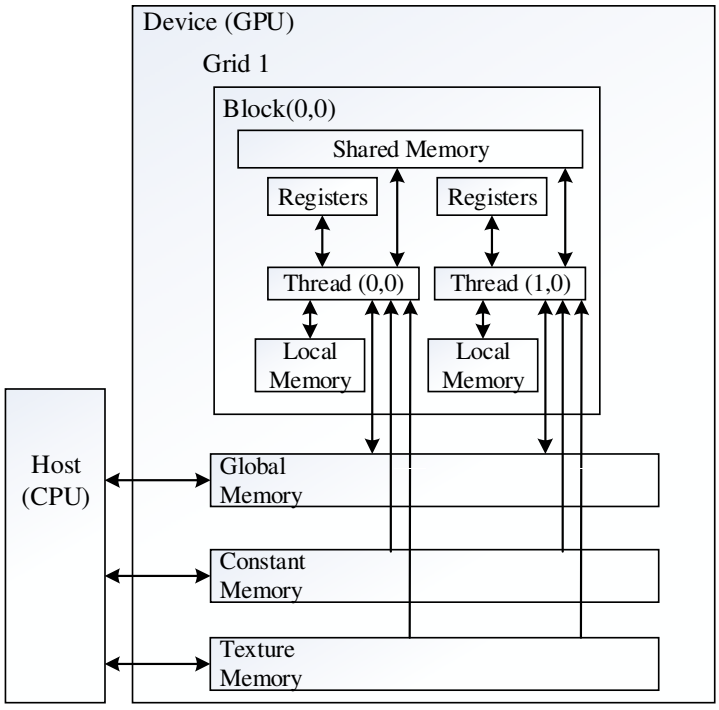


Fig. 4. The memory structure of CUDA.

The memory structure of CUDA is shown in Fig. 4. Arrows in the figure indicate data flow directions. Global memory, constant memory and texture memory are visible to all blocks and threads. In a block, each thread has its own local memory and registers, which is a fine-grained parallelism between different threads. Threads in a same block have a shared memory that enables the GPU to perform secondary parallelism, which is a coarse-grained parallelism between different blocks. It is worth noting that the read-write speed of shared memory is notably high that the threads in a same block running in parallel can synchronize with each other at different instances during the execution [32].

Blocks and threads are software concepts which are indeed allocated with specific hardware resources called streaming multiprocessors (SM) that perform actual computations. An SM is composed of multiple streaming processors (SP) [32]. The schematic diagram illustrating the SMs and SPs is shown in Fig. 5. When the program is implemented, a block is allocated to an SM to execute, and each thread in the block is allocated to an SP. 32 threads within a block are put into a warp to execute the same instruction synchronously. An SM can only execute one warp at a time.

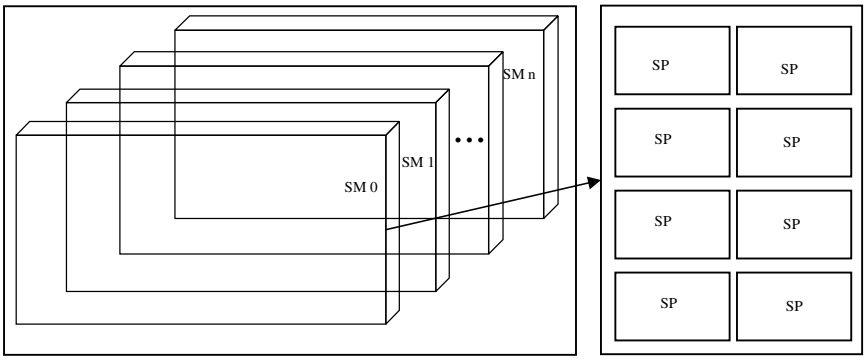


Fig. 5. The schematic diagram of GPU computing units.

3.2 Parallel learning algorithm of SDBN

Learning-based methodologies, particularly those involving neural networks, often employ parallel designs to accelerate matrix operations. However, when an evolutionary algorithm is added to the network training process, a modified design is required as it now needs to run both evolutionary algorithm and network training in parallel on GPU. In addition, it is necessary to ensure that the parallel design scheme can be easily migrated.

3.2.1. Parallel SA-QMOEA/D algorithm

The communication speed between CPU and GPU is relatively slow. In order to reduce the communication time, the parameters are usually stored in the global memory of GPU, and the input data is transferred to the GPU in batches. When using evolutionary algorithms to update parameters, there are some potential problems if we choose and reorganize individuals in a block as we don't know whether all fitness values of individuals are calculated. Therefore, a synchronization operation is required [31]. Only when all individuals complete the calculation they can be selected or reorganized. In addition, when the number of threads is less than the number of individuals, a thread needs to compute more than one individual. In order to solve this problem, we sample the population of individuals into multiple partitions as shown in Fig. 6. Consecutively, each partition is divided into various sub-populations.

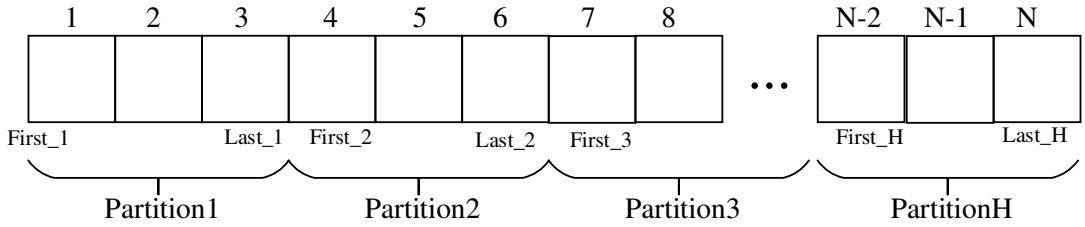


Fig. 6. The partition of population.

The population shown in Fig. 6 is divided into H partitions. “First_i” and “Last_i” are respectively the first and the last sub-populations of i^{th} partition. This partitioning strategy has two advantages: first it ensures the portability of the program, and the other is that the

multiple sub-populations can find an optimal solution more effectively. Developed SA-QMOEA/D parallel algorithm is summarized in algorithm 3.

Algorithm 3 (SA-QMOEA/D parallel algorithm)

Step 1 Initialization:

Step 1.1 Initialize N uniformly spread weight vectors $[\lambda^1,...,\lambda^N]$

Step 1.3 Initialize a field B:

- compute the Euclidean distances between any of the two weight vectors
- select T closest weight vectors $[\lambda^{i1},....\lambda^{iT}]$ for each λ^i and then get

$B(i) = (\lambda^{i1},....\lambda^{iT})$

Step 1.3 Initialize the population:

- Initialize the chromosome space $[\theta^1,...,\theta^N] \in \Omega_Q$
- transform Ω_Q to get $[x^1,...,x^N] \in \Omega$
- for each x^i in Ω: calculate $FV^i = F(x^i)$

Step 1.4 Initialization solution $z = (z_1,...,z_m)$ according to FV^i

Step 1.5 Sample the population into several sub-populations

Step 1.6 Transfer the above parameters to the global memory of GPU

Step 2 Population evolution:

for each sub-population:

for each individual i in the sub-population:

- Synchronization: Randomly select three different elements from $B(t)$ to get the corresponding quantum chromosomes. And apply the recombination and mutation operator on them to get new solution y.
- for each z_j in $(z_1,...,z_m)$: if $z_j > f_j(y)$: set $z_j = f_j(y)$.
- Synchronization: for each λ^j in B: if $g^{te}(\mathrm{y} \mid \lambda^j, \mathrm{z}) \le g^{te}(x^j \mid \lambda^j, \mathrm{z})$:

set $x^j = y$, $FV^j = F(y)$.

- Synchronization: if a solution in EP is dominated by $F(y)$: remove it

from EP; else: add $F(y)$ to EP.

Step 3 Repeat Step 2 until the termination condition is satisfied.

Step 4 Output EP

When comparing the parallel version (algorithm 3) with its counterpart (algorithm 2), the main difference lies in steps 1.5, 1.6 and 2. Step 1.5 is our partition strategy shown in Fig. 6 and step 1.6 prepare things to run the evolutionary algorithm on GPU. In Step 2, the operation of synchronization is required because the evolution is for the entire population and not for an individual.

3.2.2. Parallel learning algorithm of SRBM and SDBN

In section 2, we have introduced the forward and backward propagation formulas of SRBM. All these formulae contain matrix operations, which can be accelerated on GPU by splitting them on to multiple threads. The entire training data cannot all be stored in global memory; so, it needs to be transferred in as large chunks as possible to reduce the transfer frequency. Our overall SRBM parallel learning process is illustrated in Fig. 7 and the algorithm is summarized in algorithm 4.

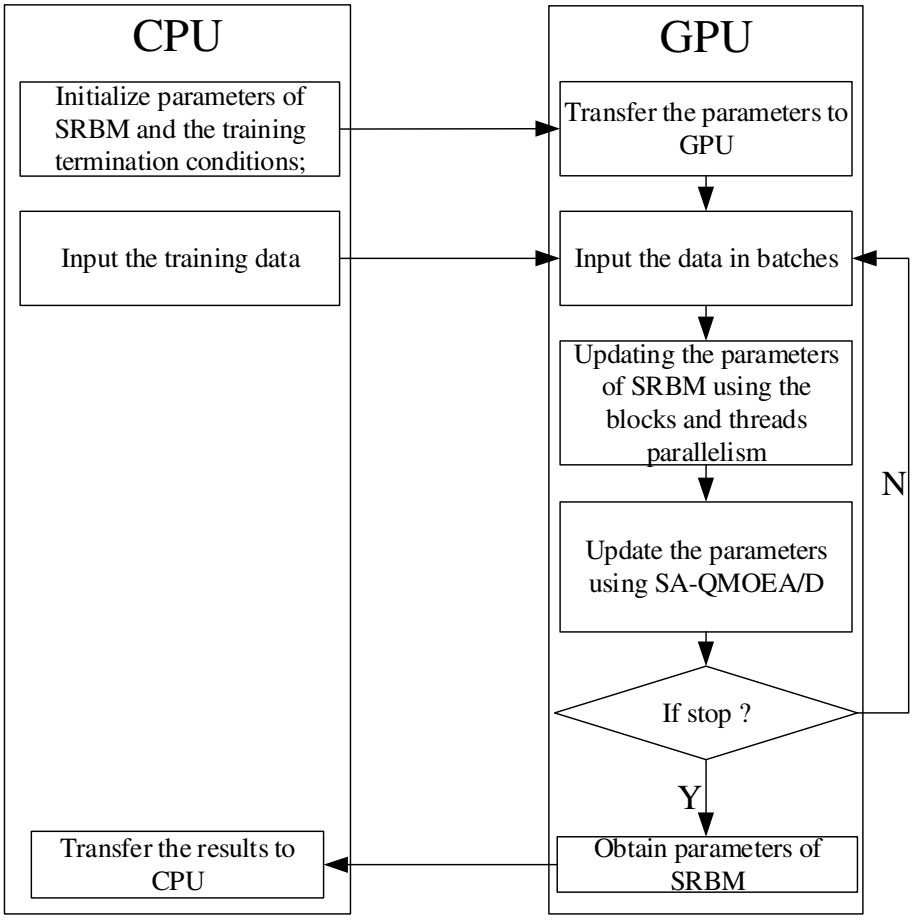


Fig. 7. The parallel learning process of SRBM.

As SDBN is composed of multiple SRBMs, it can be trained layer by layer as shown in Fig. 2 (see section 2 for details). Similarly, its training process is divided into two steps: pre-training and fine-tuning. The parallel learning algorithm of SDBN is shown in algorithm 5.

Algorithm4 (Parallel learning algorithm of SRBM based on MOP)

Step 1 Initialization:

- Initialize the parameters $\{w, a, b\}$ of SRBM
- Initialize termination conditions
- Transfer the above parameters to the global memory of GPU

Step 2 Update parameters:

Step 2.1 Transfer the training data in batches to the global memory regularly

Step 2.2 Calculate the gradients in parallel:

- Use Eq. (5) to get the gradient ∇g_{KL} of KL divergence
- Use Eq. (9) and Eq. (10) to get the gradient ∇g_{L1} of L1 regular term

Step 2.3 Update the parameters according to Eq. (11)

Step 3 Multi-objective optimization:

- Use parallel SA-QMOEA/D algorithm (algorithm 3) to get EP
- Randomly select a solution from EP as the new parameters

Step 4 Repeat Steps 2-3 until the termination condition is satisfied

Algorithm5 (Parallel learning algorithm of SDBN)

Step 1 pre-training:

Step 1.1 Using algorithm 4, train an SRBM in parallel, and its parameters are fixed after training

Step 1.2 Use the hidden layer state of the trained SRBM in Step 1.1 as the input for the next SRBM, and train the next SRBM using algorithm 4

Step 1.3 Repeat Step 1.1 and Step 1.2 until all SRBMs are trained to complete.

Step 2 fine-tuning:

Step 2.1 Use the parameters of these SRBMs obtained in Step 1 as the initial parameters of SDBN.

Step 2.2 Use BP algorithm to update SDBN network parameters in parallel until the termination condition is satisfied.

The pre-training process in algorithm 5 is based on algorithm 4. Likewise, the SDBN uses the parameters obtained in step 1 as the initial parameters of the network instead of random initialization, which can make the SDBN to converge faster while avoiding local minima.

4. Facial expression recognition algorithm based on parallel SDBN

Facial expression recognition (FER), is a challenging and prominent area of research in computer vision with a variety of applications for human-computer interaction. Main challenge lies in extracting effective features due to high intra-class variations, which raises the difficulty in developing a generalized approach [40-44]. Previous research suggests that the DBN-based methods are effective for FER where multiple RBMs are used to represent multi-level features. In this section, we show how our parallel SDBN can be used for FER. Our approach first learns abstract features in an unsupervised way through SRBM, and then performs supervised fine-tuning for effective expression recognition.

4.1 Traditional facial expression recognition method

Traditional FER methods are composed of four major steps: face detection, positioning, extracting expression features, and recognizing expressions. Most of these methods are based on manual local feature extraction and uses Gabor wavelets [36-37], local binary model (local binary pattern, LBP) [38], scale invariant feature transformation (SIFT) features [39] etc. as visual features combined with Bayesian classification, SVM, AdaBoost etc. Although the results achieved with those methods are satisfactory, they are prone to loss of related information [38-41]. Also, different illuminations and angles have a great influence on the information extracted of the face. Moreover, different people have different facial expressions. Therefore, traditional methods need to extract different features for different scenes.

While using deep neural networks for FER, feature extraction and classification steps can be combined to simplify the work [40]. At present, the neural networks used for the FER mainly include BP neural network, DBN, CNN etc. For DBN, the input must be a one-dimensional vector, so the original image needs to be preprocessed. The work-flow using a conventional DBN for FER is shown in Fig. 8.

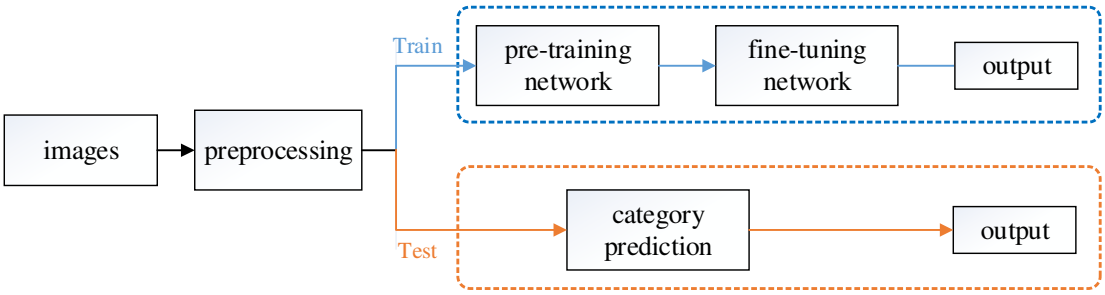


Fig. 8. Flowchart of traditional DBN for FER.

In Fig. 8, the blue line represents the training process and the yellow line represents the prediction process after training. Excessive features are not necessary for expression recognition, because expressions are usually related to the face, and are independent of scene background. Therefore, if the traditional DBN is used directly without any sparse restrictions,

the unrestricted RBM will extract a lot of redundant features. This not only increases the computation cost but also introduces a lot of noise resulting in accuracy reduction. This shortcoming can be handled effectively using our SDBN.

4.2 Facial expression recognition algorithm based on parallel SDBN

Few active hidden units can effectively extract advanced features from an image. Therefore, we use our SRBM to extract the information of the image, which can learn more effective features to achieve better recognition results. In order to generalize the algorithm, in this paper we do not perform any complex preprocessing but only reduce the dimensions of the original image by principal component analysis (PCA) [42]. The preprocessing and the process of recognizing a facial expression using our SDBN is shown in Fig. 9.

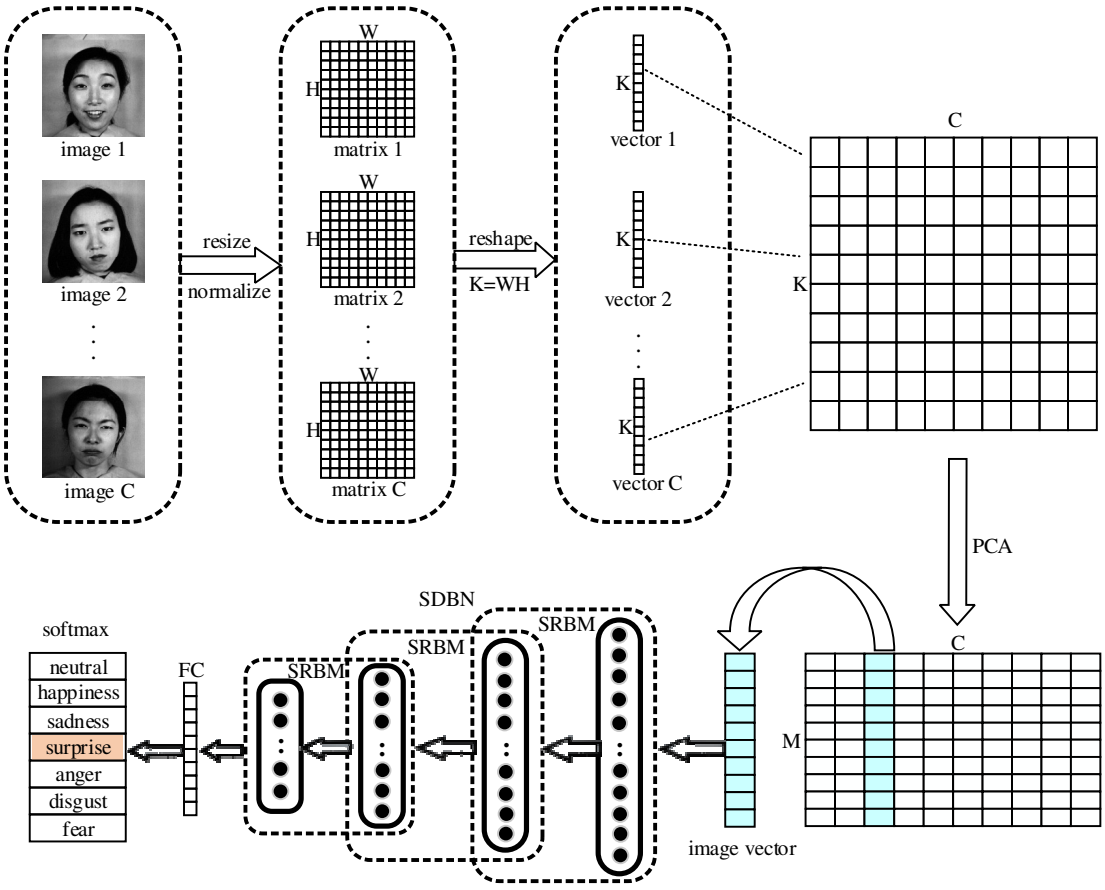


Fig. 9. The process of recognizing a facial expression using SDBN.

The preprocessing process shown in the Fig. 9 is consisted of two steps. In the first step the images are resized, normalized, and are converted into one-dimensional vectors. In the second step, these vectors are combined into a matrix which are then processed by PCA for dimensionality reduction. This step results in a new image vector, which is used as the input of SDBN. We use the hidden layer of the last SRBM as the fully connected layer and use a softmax layer at the end to perform expression recognition. These details are shown in algorithm 6.

Algorithm6 (Training process of FER algorithm based on parallel SDBN)

Step 1 Preprocessing:

Step 1.1 Obtain C training images and move them to GPU

Step 1.2 Resize and normalize the C training images. Each image is then expanded into a vector $\boldsymbol{x}_i$ with K elements. Finally, the C vectors are combined into a matrix $\mathbf{X}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_C\}$. The size of the matrix $\mathbf{X}$ is [K, C]

Step 1.3 PCA:

Step 1.3.1 For each $\boldsymbol{x}_i$ in $\mathbf{X}$: $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - \frac{1}{C}\sum_{j=1}^{C}\boldsymbol{x}_j$

Step 1.3.2 Calculate the covariance matrix cov of $\mathbf{X}$: $cov = \mathbf{X}\mathbf{X}^{\mathrm{T}}$

Step 1.3.3 Calculate the eigenvalues $\lambda_1,\lambda_2,\cdots,\lambda_K$ and eigenvectors $\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_K$ of matrix $\mathbf{X}$ by singular-value decomposition (SVD)

Step 1.3.4 Select the M largest eigenvalues and their corresponding eigenvectors $\mathbf{W}=\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_M$. The size of the matrix $\mathbf{W}$ is [K, M]

Step 1.3.5 Get the new matrix $\mathbf{X}^*$ after dimension reduction: $\mathbf{X}^*=\mathbf{W}^{\mathrm{T}}\mathbf{X}$

The size of the matrix $\mathbf{X}^*$ is [M, C], where M < K

Step 1.4 Save $\mathbf{X}^*$ to CPU.

Step 2 training SDBN:

Use the pre-processed data $\mathbf{X}^*$ and algorithm 5 to train SDBN.

In Step 1, PCA requires many matrix operations, so we use GPU for acceleration. Besides, the dataset used for the experiments is not very large, so we can move all the data to the GPU at once. However, if the dataset is large, it needs to be transferred in batches. Another point to note is that in Step 2, the SDBN uses a softmax layer as the classifier. It is only used for fine-tuning of the SDBN.

5. Experiments

In this section, we test the effectiveness of our parallel algorithm by using it for FER. The experiments are conducted on a PC with an Intel i5-6500 CPU and a Nvidia GTX950M GPU. The presented methods are implemented in Python using Pycuda library [30].

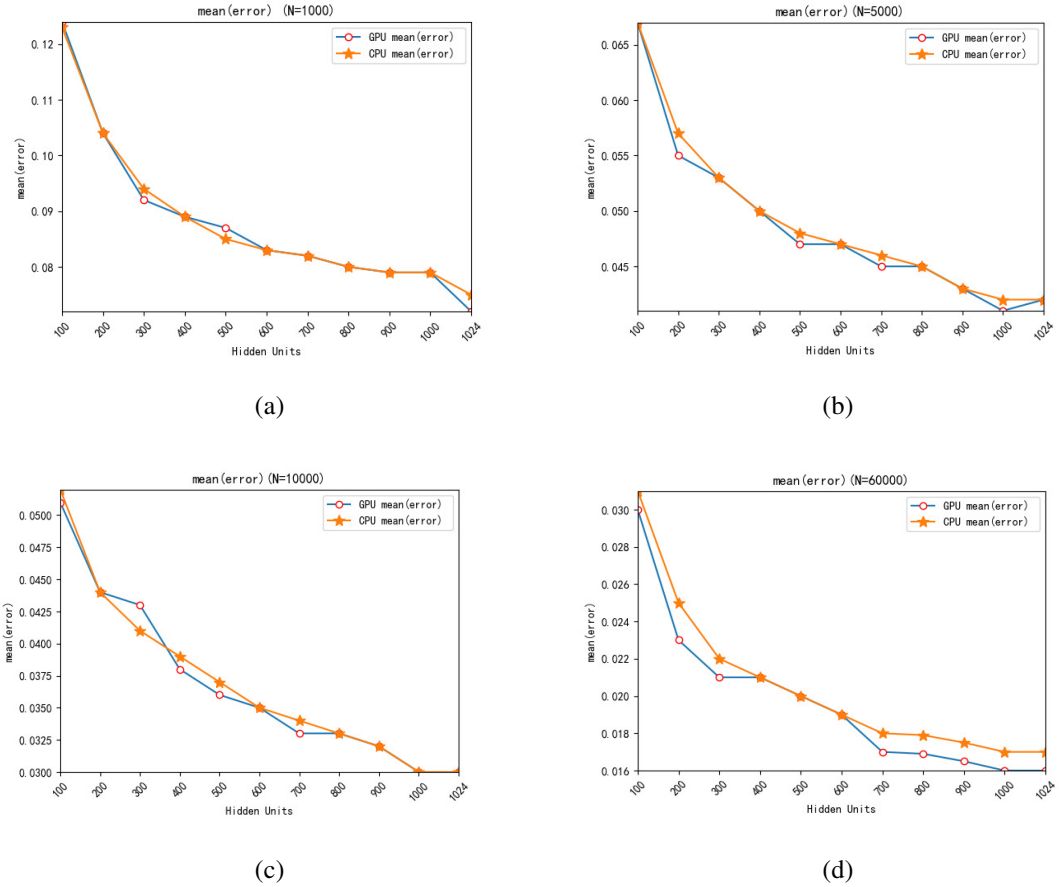
5.1 Parallel acceleration analysis

We use the MNIST dataset [33] of handwritten digits to evaluate the performance of GPU acceleration. The dataset contains 60,000 training images and 10,000 test images. Each image

is of size 28 x 28 pixels. The values of the parameters used for the experiments are as follows:

- SA-QMOEA/D algorithm: The number of individuals is 256. The number of sub-populations is 4. The number of elements in field B is 20. And the number of population iterations is 100. 4 thread blocks are allocated for the kernel, each with 64 threads.
- SRBM: The number of visible units is 784. The learning rate of SRBM is 1e-3. 96 thread blocks are allocated for the kernel, each with 512 threads.
- The batch size is 64.

Firstly, we test our SRBM algorithm on a single SRBM with different number of samples, N, and hidden units. Fig. 10 (a) – (d) show the reconstruction error for 1k, 5k, 10k, and 60k samples, respectively.



(a) (b)

(c) (d)

Fig. 10. Reconstruction error of a single SRBM with different number of samples and hidden units. The horizontal axis of each subgraph is the number of hidden units, and the vertical axis is the average reconstruction error of the SRBM after 30 epochs of training. (a) – (d) respectively show the errors for 1k, 5k, 10k, and 60k samples.

It can be seen from the figure that the performance of both CPU and GPU are reasonably same for the cases (a) and (c). However, GPU showed slightly better performance than CPU in case of (b) and (d). This is because the population is partitioned in our parallel algorithm, so it is more likely to find the optimal solution.

We calculate the speedup of the above experiment to analyze GPU acceleration. Speedup is calculated by dividing the running time of a program on the CPU by the running time on the GPU. The results are shown in Table 1 and the speedup trend is shown in Fig. 11.

Table 1. The speedup performance of a single SRBM with different number of samples and hidden units.

number of	number of hidden units										
samples	100	200	300	400	500	600	700	800	900	1000	1024
1000	1.33	1.40	1.44	1.53	1.55	1.56	1.66	1.67	1.68	1.68	**1.69**
5000	1.63	1.66	1.66	1.67	1.71	1.72	1.74	1.74	1.80	1.81	**1.82**
10000	1.82	1.86	1.86	1.87	1.87	1.97	1.97	1.97	2.01	2.01	**2.07**
60000	2.25	2.53	2.55	2.74	3.24	3.45	3.53	3.92	3.94	4.07	**4.29**

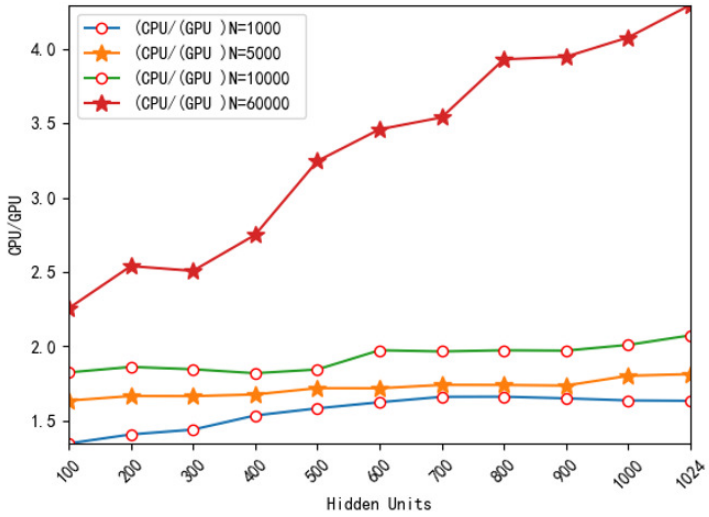


Fig. 11. The speedup trend of a single SRBM with different number of samples and hidden units.

It can be seen from Table 1 and Fig. 11 that with more hidden units and samples, the speedup is improved. And, when trained using all 60000 samples, our parallel algorithm can achieve 4 times better acceleration over CPU on a single SRBM. Furthermore, two important points can be noted here:

- In Table 1, although only a difference of 24 hidden units, speedup is higher for 1024 hidden units than for 1000 hidden units. This is since a warp contains 32 threads, when the number of hidden units is a multiple of 32, each thread can be fully utilized.
- In Fig. 11, when N=60000, the speedup is higher than other cases. This is mainly because when the sample size is small, both GPU and CPU complete the calculations rapidly; and, in this case, the time that the GPU spends reading data cannot be ignored, so the speedup of GPU is not evident. However, when the sample size is large, e.g. for N = 60k, the running time of a program depends mainly on the calculations. In this case, the data reading time can be ignored, and the advantage of

the GPU is clearly reflected.

In order to analyze the effect of number of layers and number of epochs of training, we design two SDBNs. The first SDBN is composed of 2 SRBMs, and both their hidden layers contain 1024 hidden units. The second SDBN is composed of 3 SRBMs and their hidden layers contain 1024, 1024 and 2048 hidden units, respectively. The learning rate of fine-tuning of SDBN is 1e-4. And we only take 20 epochs of fine-tuning. The training time of the two SDBNs is shown in Table 2 and Table 3.

Table 2. Training time of SDBN composed of 2 SRBMs under different sample sizes and epoch numbers. The unit of time is seconds.

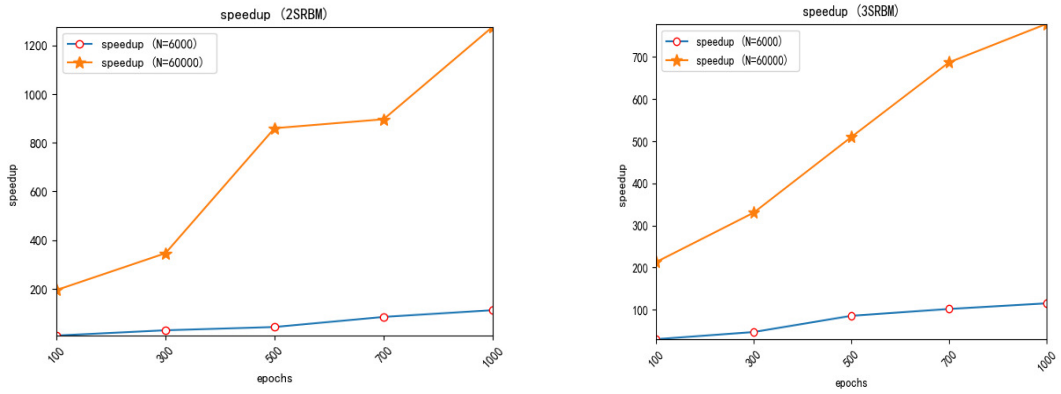
number of samples	device	number of epochs of SRBM training				
		100	300	500	700	1000
6000	CPU	753	2875	4381	8789	12183
	GPU	91	96	101	103	108
	speedup	**8.1**	**29.8**	**43.2**	**84.6**	**112.4**
60000	CPU	18619	34418	87600	94488	140231
	GPU	95	99	101	105	109
	speedup	**194.8**	**346.3**	**859.8**	**896.4**	**1276.4**

Table 3. Training time of SDBN composed of 3 SRBMs under different sample sizes and epoch numbers. The unit of time is seconds.

number of samples	device	number of epochs of SRBM training				
		100	300	500	700	1000
6000	CPU	4858	7989	15162	18823	22581
	GPU	163	171	177	185	196
	speedup	**29.6**	**46.7**	**85.2**	**101.4**	**114.8**
60000	CPU	35346	57693	93163	130672	157983
	GPU	166	174	182	190	202
	speedup	**212.6**	**330.0**	**509.7**	**686.9**	**778.5**

From the results shown in Tables 2 and 3, we can notice that with the increase of iterations, the training time of serial algorithm increases exponentially, while for the parallel algorithm it only increased by a few seconds. It also can be seen that the speedup is in the range of 8 to 1276, and the highest speedup appears in case of the SDBN containing 2 SRBMs instead of 3 SRBMs for the limitations of our computer hardware. Because, the number of threads that are contained in a block is limited by GPU. In our GPU, a block can only contain up to 1024 threads. However, the number of hidden units in the last layer of the SDBN composed of 3 SRBMs is 2048. Therefore, the training time of the SDBN with 3 SRBMs increases rapidly compared to the SDBN with 2 SRBMs. Although the speedup of the SDBN composed of 3

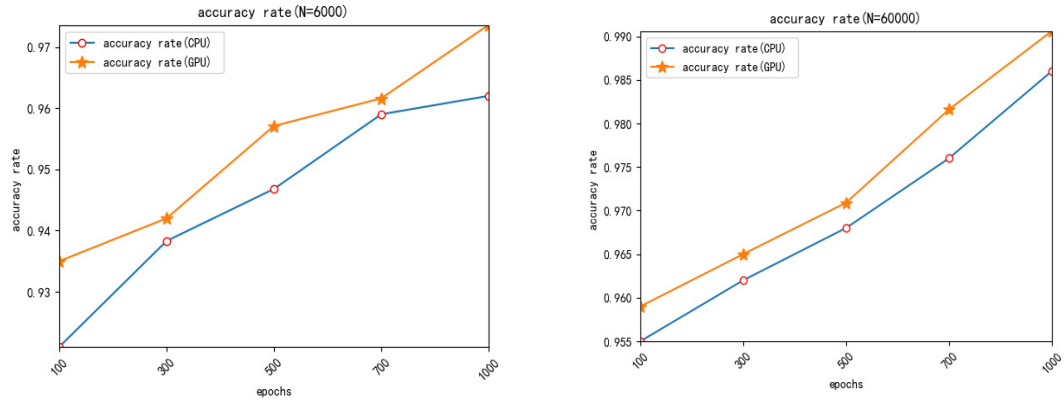
SRBMs is lower than that of 2 SRBMs, it still achieves a good acceleration effect with a maximum speedup of 778. The trend followed by the speedup is shown in Fig. 12.



(a) (b)

Fig. 12. The trend of speedup under different number of samples and epochs.

From Fig. 12, we can see that, in both cases, the speedup is higher for 60000 samples (yellow line with stars) than 6000 samples (blue line with circles). This means that the larger the sample size, the more evident the GPU advantage. Simultaneously, we calculate the classification accuracy of these two SDBNs on the MNIST dataset. The results are shown in Fig. 13 and Fig. 14.



(a) (b)

Fig. 13. The accuracy rate of SDBN composed of 2 SRBMs.

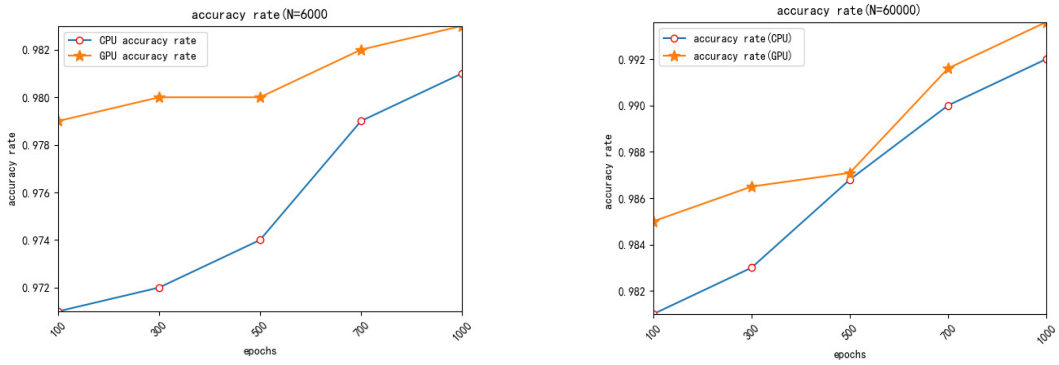


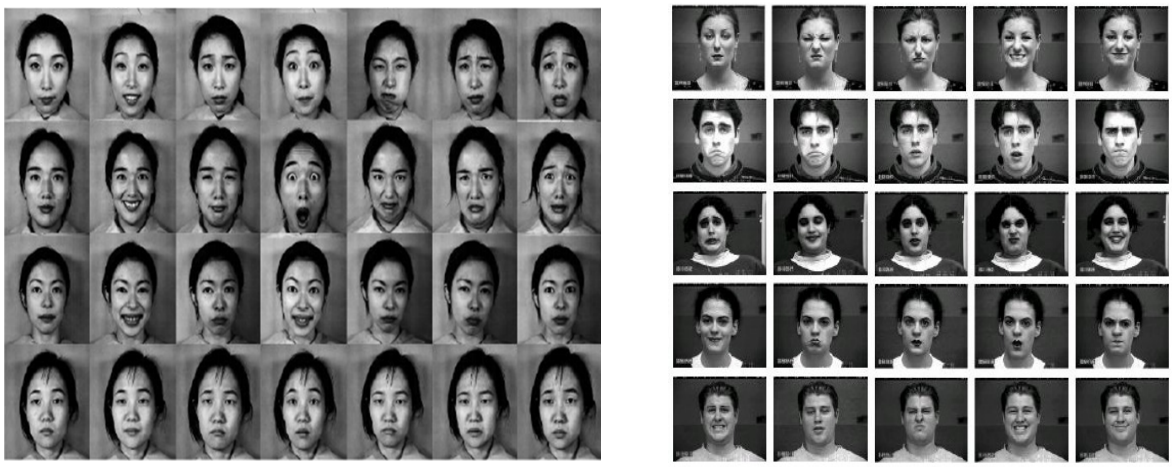
Fig. 14. The accuracy rate of SDBN composed of 3 SRBMs.

From these results, we can see that the accuracy is increased with the number of samples and epochs. Also, parallel algorithm performs better than the serial one. This is because the sub-populations in our parallel algorithm can converge to a better optimal solution. The highest accuracy of our model is over 99%, which shows that our algorithm can learn abstract features to achieve better classification performance with significantly lower training time.

All the above experimental results demonstrate the superiority of our parallel algorithm of SRBM and SRBN in terms of acceleration and accuracy performance. In the following section we study the performance of our method by applying it to perform facial expression recognition.

5.2 Facial expression recognition analysis

For these experiments, we use the publicly available JAFFE [43-44] and CK+ [45] datasets to verify the effectiveness of our algorithm. JAFFE is a small dataset with 213 images, with an image size of 256 x 256 pixels. While CK+ is a large dataset with 2100 images, each of which has a size of 640 x 490 pixels. Sample images from each dataset are shown in Fig. 15.



(a) (b)

Fig. 15. Examples of the two datasets. (a): JAFFE dataset. (b): CK+ dataset.

As discussed in Section 4, if the original image size is directly used as the input of SDBN, it increases the computation cost. For example, using an image of size 640 x 490 pixels, the number of units in the visual layer should be 313600, which is extremely expensive to calculate and makes it difficult for SRBM to learn effective features. Therefore, we first preprocess the dataset according to the method described in Section 4. The following parameters are used for the experiments:

- The SDBN used in the following experiments is composed of 3 SRBMs, and their hidden layers contain 1024, 1024 and 2048 hidden units, respectively. The number of units in SDBN visual layer is the size of the resultant image vector after preprocessing.
- SA-QMOEA/D algorithm: The number of individuals is 256. The number of sub-populations is 4. The number of the field B is 20, and the number of population iterations is 100. This kernel is allocated 4 blocks, each with 64 threads.
- The learning rate of SRBM is 1e-3. The initial learning rate of fine-tuning for training SDBN is 1e-4, but if the loss does not decrease after 5 epochs, the learning rate of fine-tuning will be halved. And the maximum number of training epochs is 30. This kernel is allocated 96 blocks, each with 512 threads. And the batch size is 64.

5.2.1. Facial expression recognition using JAFFE dataset

In this subsection, we use the small JAFFE dataset to verify the effectiveness of our algorithm. We use three different image vectors whose dimensions are 30x36, 60x72, and 90x108. Note that here we use 30x36 instead of 1080 to represent the size of an image vector, which is more in line with the image size. The accuracy rate of our SDBN with different dimensions and different training epochs is shown in Table 4, and its accuracy rate trend is shown in Fig. 16.

Table 4. Accuracy rate of SDBN with different image sizes and number of epochs in JAFFE dataset.

image	number of epochs of SRBM training									
size	100	200	300	400	500	600	700	800	900	1000
30x36	0.33	0.59	0.65	0.71	0.79	0.82	0.84	0.86	0.89	**0.91**
60x72	0.57	0.63	0.77	0.88	0.90	0.92	0.94	0.97	0.98	**0.99**
90x108	0.585	0.684	0.752	0.78	0.885	0.95	0.965	0.98	0.99	**0.992**

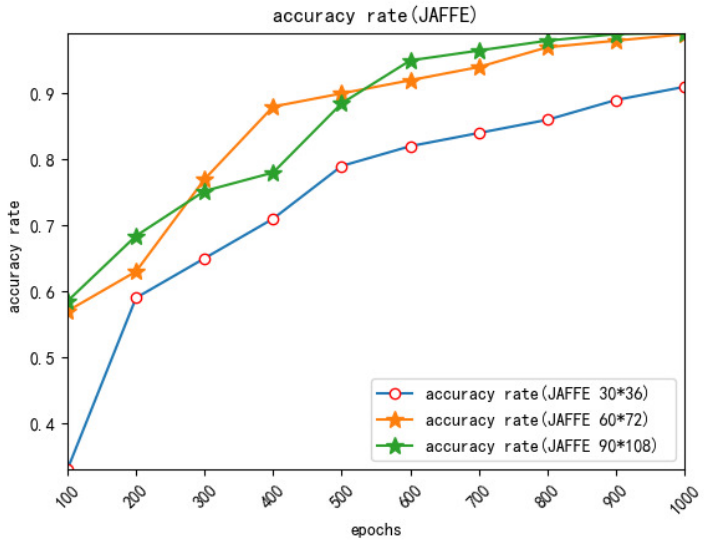


Fig. 16. Accuracy rate trend of SDBN with different image sizes and number of epochs in JAFFE dataset.

Form Table 4 we can see that with the increase of epochs, the FER accuracy rate increases and reaches its highest at 1000. The lower the image size, the less information they contain, so the accuracy rate is low. But the image size cannot be increased indefinitely, because it requires additional computations and storage. From Fig. 16 we can see that 60x72 is a better choice for the image size because it requires less computation to achieve higher FER accuracy.

Next, we use the SDBN with the best results achieved and compare its performance against the conventional methods like nearest neighbor, support vector machine, sparse representation classification and traditional DBN algorithm. The comparison results are shown in Table 5. It can be seen from the results that our method outperformed all the conventional methods for all the image sizes.

Table 5. Comparison of accuracy rate with different methods using JAFFE dataset.

image size	method				
	NN	SVM	SRC	DBN	SDBN
30x36	78.75	90.63	83.13	90.25	**91.12**
60x72	86.25	91.88	89.38	98.75	**99.00**
90x108	88.65	94.48	90.38	98.95	**99.20**

5.2.2. Facial expression recognition using CK+ dataset

As a further validation of our method, we have conducted more experiments using CK+ dataset. We process the image into three different dimensions: 24x24, 32x32 and 64x49. And the number of epochs of SRBM training is 800. We explore the effect of the number of hidden units on the accuracy rate. The obtained results are shown in Fig. 17.

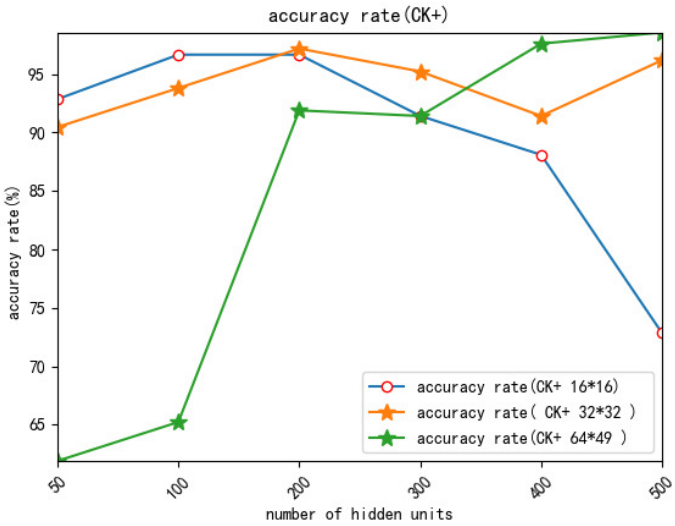


Fig. 17. Accuracy rate of SDBN with different number of hidden units and image size in CK+ datasets.

As can be seen from the figure, at the beginning, the accuracy rate increased with the number of hidden units up to an extent and started to fall with further increase in the number of hidden units. The lower the image size, the fewer the hidden units are needed to achieve the highest accuracy rate, which means the network structure that needs to interpret the data is simpler when the image size is relatively small. The comparison results between our SDBN and other methods using CK+ dataset are shown in Table 6. Even in this case our method outperformed the rest. Unlike sudden change of accuracy rate in case of NN and SVM, our method exhibited stable performance.

Table 6. Comparison of FER accuracy rate with different methods in CK+ datasets.

image size	method				
	NN	SVM	SRC	DBN	SDBN
24x24	95.71	65.23	97.14	97.28	**97.52**
32x32	74.28	98.09	97.10	97.75	**98.30**
64x49	88.65	95.48	98.58	98.57	**98.90**

All the experimental results of FER show that our method has high recognition accuracy rate and stable performance.

5.2.3. Train time and inference time

In order to further illustrate the superiority of our algorithm in terms of speed, we have recorded the training time and inference time of all the methods used for FER. Results are summarized in Tables 7 and 8.

Table 7. Training time and inference time in seconds for different methods in JAFFE dataset. '-' indicates that there is no training process.

time	image size	method					
		NN	SVM	SRC	DBN	SDBN (CPU)	SDBN (GPU)
training	30x36	-	**146.2**	-	672.3	2734.6	184.1
	60x72	-	683.5	-	1419.7	5068.2	**197.0**
	90x108	-	1342.2	-	2378.4	8624.3	**215.8**
inference	30x36	4.6	0.3	42.1	7.7	6.2	**0.3**
	60x72	17.7	0.4	106.2	15.7	12.4	**0.3**
	90x108	37.4	0.6	314.9	24.5	21.7	**0.5**

Table 8. Training time and inference time in seconds for different methods in CK+ dataset. '-' indicates that there is no training process.

time	image size	method					
		NN	SVM	SRC	DBN	SDBN (CPU)	SDBN (GPU)
training	24x24	-	973.3	-	5396.7	21072.1	**282.4**
	32x32	-	1727.8	-	6021.5	24396.3	**294.3**
	64x49	-	5124.1	-	9614.4	37323.0	**331.6**
inference	24x24	1135.2	**1.4**	321.2	54.7	48.9	1.7
	32x32	1996.7	1.8	416.7	78.9	56.7	**1.8**
	64*49	6102.3	**2.1**	757.4	126.3	86.4	2.2

The following conclusions can be drawn from these results:

- Our parallel SDBN has more advantages in terms of training time and inference time, and it does not change dramatically with the increase of data dimensions. But the inference time of NN and the training time of SVM increase rapidly with the increase of data dimensions and the number of images (CK+ dataset is larger than JAFFE dataset).
- Because our SDBN uses evolutionary algorithm to optimize, it takes longer training time without parallel implementation. While using parallel SDBN (with GPU) the training time is short.
- On CK+ dataset, SDBN (with GPU) inference time is slightly slower than SVM. There are two main reasons. First, the LIBSVM library [48] has been greatly optimized for computational speed. Second, SDBN (with GPU) inference time is limited by the batch size. If GPU memory increases, we can use larger batch sizes to further reduce the inference time. It is worth noting that the SVM is generally more suitable for small datasets, and its classification accuracy is not as good as our SDBN for larger ones.

- In inference time, our SDBN (CPU) is slightly faster than DBN because of the sparsity of our SDBN.

6. Conclusion

In this paper, we have elaborated the principles and advantages of SDBN. In our SDBN, we construct a multi-objective optimization problem from RBM, and use the evolutionary algorithm to solve the problem, which is an alternate way of combining evolutionary algorithms and neural networks. Our multi-objective optimization algorithm can automatically search a set of optimal solutions for the sparse penalty term in RBM, which can achieve the balance between over-fitting and under-fitting, and achieve better results. Next, we have proposed an efficient parallel algorithm to overcome the weak points that are difficult to train. We have not only realized the parallelism of the DBN network, but also have realized the parallelism of the multi-objective optimization algorithm. We have implemented the parallel design of SDBN on the GPU. Obtained experimental results show that our parallel implementation on GPU achieves a speedup performance ranging from 8 to 1200 with a higher accuracy rate when compared to that on CPU. We have also compared the training time of our sparse DBN with different number of samples, hidden units and hidden layers. Later, we analyzed the performance of our parallel SDBN by applying it to the problem of facial expression recognition and by comparing its results to that of the conventional methods using two publicly available datasets, JAFFE and CK+. The obtained results clearly demonstrated the efficiency of our approach. In future, we will study the application of SDBN to a wider range of problems, such as video, audio, text and other higher dimensional data problems.

Compliance with Ethical Standards

Funding: This work was supported by the National Natural Science Foundation of China under Grant 61772399, Grant U1701267, Grant 61773304, Grant 61672405 and Grant 61772400, the Technology Foundation for Selected Overseas Chinese Scholar in Shaanxi (Nos. 2017021 and 2018021), the Program for Cheung Kong Scholars and Innovative Research Team in University Grant IRT_15R53, the Fund for Foreign Scholars in University Research and Teaching Programs (the 111 Project) Grant B07048, the Major Research Plan of the National Natural Science Foundation of China Grant 91438201, and the Key Research and Development Plan of Innovation Chain of Industries in Shaanxi Province under Grant 2019ZDLGY09-05.

Conflict of Interest: Yangyang Li, Shuangkang Fang, Xiaoyu Bai, Licheng Jiao, and Naresh Marturi declare that they no conflict of interest.

Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent: Informed consent was obtained from all individual participants included in the study.

Reference

[1] Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets[J]. Neural computation, 2006, 18(7): 1527-1554.

[2] Hinton G E, Sejnowski T J. Learning and relearning in Boltzmann machines[J]. Parallel distributed processing: Explorations in the microstructure of cognition, 1986, 1(282-317): 2.

[3] Geng Z, Li Z, Han Y. A new deep belief network based on RBM with glial chains[J]. Information Sciences, 2018, 463: 294-306.

[4] Chen L, Zhou M, Su W, et al. Softmax regression based deep sparse autoencoder network for facial emotion recognition in human-robot interaction[J]. Information Sciences, 2018, 428: 49-61.

[5] Ranzato M A, Boureau Y L, Cun Y L. Sparse feature learning for deep belief networks[C]//Advances in neural information processing systems. 2008: 1185-1192.

[6] Zheng P, Zhao Z Q, Gao J, et al. A set-level joint sparse representation for image set classification[J]. Information Sciences, 2018, 448: 75-90.

[7] Ji N N, Zhang J S, Zhang C X. A sparse-response deep belief network based on rate distortion theory[J]. Pattern Recognition, 2014, 47(9): 3179-3191.

[8] Keyvanrad M A, Homayounpour M M. Normal sparse deep belief network[C]//2015 international joint conference on neural networks (IJCNN). IEEE, 2015: 1-7.

[9] Gong M, Liu J, Li H, et al. A Multiobjective Sparse Feature Learning Model for Deep Neural Networks[J]. IEEE Transactions on Neural Networks & Learning Systems, 2015, 26(12):3263-3277.

[10] Li Y, Bai X, Liang X, et al. Sparse Restricted Boltzmann Machine Based on Multiobjective Optimization[C]//Asia-Pacific Conference on Simulated Evolution and Learning. Springer, Cham, 2017: 899-910.

[11] Hinton G E. Training products of experts by minimizing contrastive divergence[J]. Neural computation, 2002, 14(8): 1771-1800.

[12] Banach S. Theory of linear operations[M]. Elsevier, 1987.

[13] Davis G, Mallat S, Avellaneda M. Adaptive greedy approximations[J]. Constructive approximation, 1997, 13(1): 57-98.

[14] Ng A Y. Feature selection, L 1 vs. L 2 regularization, and rotational invariance[C]//Proceedings of the twenty-first international conference on Machine learning. ACM, 2004: 78.

[15] Lee H, Battle A, Raina R, et al. Efficient sparse coding algorithms[C]// International Conference on Neural Information Processing Systems. MIT Press, 2006:801-808.

[16] Deb K, Pratap A, Agarwal S, et al. A fast and elitist multiobjective genetic algorithm: NSGA-II[J]. IEEE Transactions on Evolutionary Computation, 2002, 6(2):182-197.
[17] Zhang Q, Li H. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition[J]. IEEE Transactions on Evolutionary Computation, 2008, 11(6):712-731.
[18] Franco M A, Bacardit J. Large-scale experimental evaluation of GPU strategies for evolutionary machine learning[J]. Information Sciences, 2016, 330: 385-402.
[19] Zhang J, Zhu Y, Pan Y, et al. Efficient parallel boolean matrix based algorithms for computing composite rough set approximations[J]. Information Sciences, 2016, 329: 287-302.
[20] Li T, Dou Y, Jiang J, et al. Optimized deep belief networks on CUDA GPUs[C]// International Joint Conference on Neural Networks. IEEE, 2015:1-8.
[21] Ly D L, Chow P. A high-performance fpga architecture for restricted boltzmann machines[C]//Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2009: 73-82.
[22] Zhang K, Chen X W. Large-Scale Deep Belief Nets with MapReduce[J]. Access IEEE, 2015, 2(2):395-403.
[23] Aarts E, Korst J. Simulated annealing and boltzmann machines[J]. Handbook of Brain Theory & Neural Networks, 1989.
[24] Bengio Y. Learning deep architectures for AI[J]. Foundations and trends® in Machine Learning, 2009, 2(1): 1-127.
[25] Hinton G E. A practical guide to training restricted Boltzmann machines[M]//Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012: 599-619.
[26] Carreira-Perpinan M A, Hinton G E. On contrastive divergence learning[C]//Aistats. 2005, 10: 33-40.
[27] Hinton G E. Training products of experts by minimizing contrastive divergence[J]. Neural computation, 2002, 14(8): 1771-1800.
[28] Yang S Y, Jiao L C, Liu F. The Quantum Evolutionary Algorithm[J]. Chinese Journal of Engineering Mathematics, 2006, 23(2):235-246.
[29] Venske S M S, Gonçalves R A, Delgado M R. ADEMO/D: Adaptive Differential Evolution for Multiobjective Problems[C]//Brazilian Symposium on Neural Networks. IEEE, 2012:226-231.
[30] Klöckner A, Pinto N, Lee Y, et al. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing[J]. Parallel Computing, 2009, 38(3):157-174.
[31] Cook S. CUDA programming: a developer's guide to parallel computing with GPUs[M]. Newnes, 2012.
[32] Xu Y, Rui W, Goswami N, et al. Software Transactional Memory for GPU Architectures[J]. IEEE Computer Architecture Letters, 2017, 13(1):49-52.

[33] LeCun Y. The MNIST database of handwritten digits[J]. http://yann. lecun. com/exdb/mnist/, 1998.

[34] Bengio Y. Learning Deep Architectures for AI[J]. Foundations & Trends® in Machine Learning, 2009, 2(1):1-127.

[35] Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors[J]. Nature, 1986, 323(6088):533-536.

[36] Tian Y, Kanade T, Cohn J F. Evaluation of Gabor-wavelet-based facial action unit recognition in image sequences of increasing complexity[C]//Proceedings of Fifth IEEE International Conference on Automatic Face Gesture Recognition. IEEE, 2002: 229-234.

[37] Lyons M, Akamatsu S, Kamachi M, et al. Coding facial expressions with gabor wavelets[C]//Proceedings Third IEEE international conference on automatic face and gesture recognition. IEEE, 1998: 200-205.

[38] Ojala T, Pietikäinen M, Mäenpää T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2002 (7): 971-987.

[39] Lowe D G. Distinctive image features from scale-invariant keypoints[J]. International journal of computer vision, 2004, 60(2): 91-110.

[40] Ranzato M, Susskind J, Mnih V, et al. On deep generative models with applications to recognition[C]//Computer Vision and Pattern Recognition. IEEE, 2011:2857-2864.

[41] Rifai S, Bengio Y, Courville A, et al. Disentangling factors of variation for facial expression recognition[C]//European Conference on Computer Vision. Springer, Berlin, Heidelberg, 2012: 808-822.

[42] Wold S, Esbensen K, Geladi P. Principal component analysis[J]. Chemometrics and intelligent laboratory systems, 1987, 2(1-3): 37-52.

[43] Lyons M J, Akamatsu S, Kamachi M, et al. The Japanese female facial expression (JAFFE) database[C]//Proceedings of third international conference on automatic face and gesture recognition. 1998: 14-16.

[44] Lyons M J, Budynek J, Akamatsu S. Automatic classification of single facial images[J]. IEEE transactions on pattern analysis and machine intelligence, 1999, 21(12): 1357-1362.

[45] Lucey P, Cohn J F, Kanade T, et al. The Extended Cohn-Kanade Dataset (CK+): A complete dataset for action unit and emotion-specified expression[C]// Computer Vision and Pattern Recognition Workshops. IEEE, 2010:94-101.

[46] Cover T M, Hart P. Nearest neighbor pattern classification[J]. IEEE transactions on information theory, 1967, 13(1): 21-27.

[47] Suykens J A K, Vandewalle J. Least squares support vector machine classifiers[J]. Neural processing letters, 1999, 9(3): 293-300.

[48] Chang C C, Lin C J. LIBSVM: A library for support vector machines[J]. ACM transactions on intelligent systems and technology (TIST), 2011, 2(3): 27.

[49] Wright J, Yang A Y, Ganesh A, et al. Robust face recognition via sparse representation[J].

IEEE transactions on pattern analysis and machine intelligence, 2008, 31(2): 210-227.